Code Generation and Optimization
Sometimes compilers do not generate machine code

• Turbo Pascal produced “byte codes” that were interpreted.
  – Portable
  – Small footprint
  – Fast compile and test cycles

• javac produces java byte code for interpretation for *dynamic compilation* (also just in time compiling, or *jit*ing)

• Microsoft’s .NET uses CIL (common intermediate language) generated by Visual Studio front-ends to be dynamically compiled.
  – CIL and javac motivated by portability and (in the CIL case) language interoperability.
Assembly Code Generation

- A simple code generation approach:

  macro-expansion of IR tuples

  Each tuple produces code independently of its context:

  - **advantage**: simple, straightforward, easy to debug
  - **disadvantage**: no optimization

    E.g., 
    
    \[(+,a,b,c)\]
    
    generates store C
    
    \[(+c,d,e)\]
    
    generates a (redundant) load C

    Peephole optimizations help a little
Peephole Optimizations

• Simple pattern-match optimizations usually following a simple code generator.
  e.g., pattern: store R X, followed by load R X
  → delete load R X

• Can recognize patterns that can be performed by special instructions (machine-specific).
  – MOVE A R_i ; MOVE A+1 R_{i+1} \Rightarrow
  – Replace by DBLMOVE A R_i
  – PowerPC Load/Store Quad Word an example of this
Peephole Optimizations

- **Constant folding:**
  - ADD lit1 lit2 result ⇒ MOVE lit1+lit2 result
  - MOVE lit1 res1 ⇒ MOVE lit1 res1
    ADD lit2 res1 res2 ⇒ MOVE lit1+lit2 res2

- **Strength reduction**
  - MUL op 2 res ⇒ SHIFTL op 1 res
  - DIV op 4 res ⇒ SHIFTR op 2 res

- **Null sequences**
  - ADD op 0 res ⇒ MOVE op res
  - MUL op 1 res ⇒ MOVE op res

- **Combine operations**
  - MOVE A R_i ; MOVE A+1 R_{i+1} ⇒ DBLMOVE A R_i
  - JEQ L1 ; JMP L2 ; L1: ⇒ JNE L2
More Peephole Optimizations

- Simplify by algebraic laws
  - $\text{ADD lit op res} \Rightarrow \text{ADD op lit res (add immediate)}$
  - $\text{SUB op 0 res} \Rightarrow \text{NEG op res}$

- Special case instructions (or why C has ++/-- operator)
  - $\text{SUB 1 R} \Rightarrow \text{DEC R}$
  - $\text{ADD 1 R} \Rightarrow \text{INC R}$
  - $\text{MOVE 0 R ; MOVE R A} \Rightarrow \text{CLR A or mvi 0 A}$

- Address mode operations
  - $\text{MOVE A R1 ; ADD 0(R1) R2} \Rightarrow \text{ADD @A R2}$
Better Schemes

• Keep “state” information

  an input IR tuple just changes the state. Code is generated as necessary when the machine changes state

• Generate code for an IR subtree all at once

• Template matching, code generation for entire template
Code Generation steps

• 4 steps:
  – instruction selection
    • this is very machine-specific. Some machines may provide complex instructions that perform 2 or more tuple operations.
  – address mode selection - easier with RISC, harder with CISC/Intel
  – register allocation
  – code scheduling (*not in text book*)

in reality, these tasks are intertwined
Address Mode Selection

- Even a simple instruction may have a large set of possible address modes and combinations. For example:
- \((+, a, b, c)\)
  - can be indexed, indirect, live register, unassigned register, memory address
  - can be literal, indexed, live register, dead register, memory address

There are more than 100 combinations
More Choices for Address Mode

- Auto increment, decrement (especially in embedded)
- Three-address instructions
- Distinct address and data registers (X86/IA32)
- Specialized registers (cond. codes, fp, index, accumulator)
- “Free” addition in indexed mode:
  
  MOVE (Reg1)offset Reg2
  
  (This is very useful for subscript operations)
Before we proceed with Code Generation:

Two important issues/optimizations that affect code generation:

– Common subexpression elimination
  • An optimization that can simplify the generated code significantly

– Aliasing
  • An issue that complicates code generation
Common Subexpression Elimination

(remove redundant computation)

1: \( A = B + C \times D \)  
keep the result in a temporary and reuse for stmt 2

2: \( E = B + C \times D \)

Difficulty: recognize when the expression is “killed”

1: \( A = B + C \times D \)
   \( B = \) <new value>  
2: \( E = B + C \times D \)

B is killed. The expression it held is no longer “(a)live”.

Easier local (i.e. within a basic block) problem than a global problem.

Pointers make it harder, i.e. what if \( B = <\text{new value}> \) was \(*X = <\text{new value}>?\)

Either “\(*X = \)" kills B, or compiler proves value \( X \neq \text{addr}(B)\)
Value Numbering

Give unique numbers to **expressions** that are computed from the same operands \( X \)

Operands are the same if

1. Their name is the same
2. They were not modified since the previous expression was computed

The Value Numbering compiler algorithm keeps a “last defined” attribute for every variable and for every temporary holding an expression.

**The Use of Value Numbering in CSE is obvious:**

A temporary holding the result of \( \text{expression}_1 \) can be reused in place of \( \text{expression}_2 \) if the two expressions have the same value number.
Avoid/Remove Redundant Computation

• We use recognized common subexpressions to
  – suppress code generation for such subexpressions and use the temporary holding the needed value instead, or
  – replace the (already generated code of the) subexpression with the temporary.

Proper bookkeeping is necessary to mark the temporaries as alive and still needed.
Do tmp? and tmp?? Contain the same value? I.e. is it ok to write:

\[ A = B + C \]

\[ Y = B + C \]

\[ +, B, C, \text{tmp} \]

\[ \text{MOVE} \ \text{tmp}, A \]

\[ \ldots \]

\[ +, B, C, \text{tmp}?? \]

\[ \text{MOVE} \ \text{tmp??}, Y \]

And then

\[ +, B, C, \text{tmp1} \]

\[ \text{MOVE} \ \text{tmp1}, A \]

\[ \ldots \]

\[ +, B, C, \text{tmp2} \]

\[ \text{MOVE} \ \text{tmp2}, Y \]

\[ \text{MOVE} \ \text{tmp1}, Y \]
Local CSE Example

Do tmp? and tmp?? Contain the same value? I.e. is it ok to write:

+ B, C, tmp1
MOVE tmp1, A
...
+ B, C, tmp1
MOVE tmp1, Y

And then

+ B, C, tmp1
MOVE tmp1, A
...
MOVE tmp1, Y

If \texttt{operation1} = \texttt{operation2} AND B = B AND C = C AND \texttt{Last_def[C]} < \texttt{last_def[tmp1]} && \texttt{last_def[B]} < \texttt{last_def[tmp1]}

How do we detect the two tmps created by the + operations have the same value?
CSE: Detecting Equivalent Expressions

• Possible method: create a name for the temporary that is derived from the operators and operands in a unique way.

\[
\begin{align*}
B &= X + Y \cdot Z \quad \text{temp name: X\_p\_Y\_t\_Z} \\
X &= X + Y - Z \quad \text{temp name: X\_p\_Y\_m\_Z} \\
D &= X + Y \cdot Z \quad \text{temp name: X\_p\_Y\_t\_Z}
\end{align*}
\]

*Static single assignment* (SSA) is one way to represent operand values as a name. We will not cover SSA in this course in detail.
Alias Problem

• A big problem in compiler optimizations is to recognize *aliases*.
• Aliases are “different names for the same storage location”
• Aliases can occur in the following situations
  – pointers may refer to the same variable
  – arrays may reference the same element
  – Subroutine calls may pass in the same variable under two different names
  – subroutines may have side effects
  – Explicit storage overlapping (unions, FORTRAN equivalence and common)
• The ramification here is, that we cannot be sure that variables hold the values they appear to hold. We need to conservatively mark values as *killed*. 
main( ) {
    int *A, *B, i, j;
    int C[100], D[100];
    If (fee( )) {
        A = C;
        B = D;
    } else {
        A = C;
        B = C;
    }
    j = fie( );
    for (i=0; i <100; i++)
}

If the **true** branch is taken, **A** and **B** point to different storage.

If the **else/false** branch is taken, then **A** and **B** point to the same storage.

In the **for** loop, we must generate code so that the write to **A[i]** does not clobber value of **B[i]** before it is used, and that the right value of **B[j]** is read. What the compiler generates must be correct regardless of the **if** branch taken and the value of **j**.
The call \texttt{foe(x,x)} produces aliased parameters \texttt{a} and \texttt{b}.

The call \texttt{foe(x,y)} produces un-aliased parameters \texttt{a} and \texttt{b}.

The code produced for foe must be correct in both cases (or can do “cloning”, but that’s beyond where we are now.)
General problem of “memory disambiguation”

• It has been said that all of compiler analysis reduces to memory disambiguation
  – *Memory disambiguation* is the problem of deciding if two references (e.g. to *a* and *b* in *foe*) are references to the same location
  – If different location referenced, the references are *independent* and they do not have to be ordered relative to one another
  – Otherwise *dependent*, and the order must be maintained.
Global and local optimizations

• **Local optimizations**: operation is within a *basic block* (BB).
  – A BB is a section of code without branches (except possibly at the end) and no labels, except possibly at the beginning
  – BBs can be from a few instructions to several hundred instructions long.
  – Execution of any instruction in BB implies execution of them all

• **Global optimizations** (across an entire procedure, often called *intraprocedural*) will be introduced later.

• **Interprocedural** optimization/analysis used to refer to the “whole program” case. Also sometimes referred to as global optimization. Will discuss some later.
Register Allocation Issues

• 1. Eliminate register loads and stores
  
  store R3,A
  
  ... we want to recognize that R3 could be reused
  
  load R4,A

• 2. Reduce register spilling.
  
  – Ideally all data is kept in registers until the end of the BB. However, there may not be enough registers. What registers should be freed (by writing, or spilling values, to memory)?

Optimal solutions are NP-complete problems
Top-Down Register Allocation  
(Cooper/Torczon p. 385)

• Basic idea:
  In each basic block (BB) do this:
  – find the number of references to each variable
  – assign available registers to variables with the most references
Details:
  – keep some free registers for operations on unassigned variables
  – store *dirty* registers at the end of the BB. Do this only for variables (not for temporaries )
  • not doing this for temporaries exploits the fact that they are never live-out of a block. This is knowledge that would otherwise need global analysis.
Bottom-Up Register Allocation
(Cooper/Torczon p. 386)

for each tuple \( op \ A \ B \ C \) in a BB do :

\[
\begin{align*}
& r_x = \text{ensure}(A) \quad \text{// make sure A is in a register} \\
& r_y = \text{ensure}(B) \quad \text{// make sure B is in a register} \\
& \text{if } r_x \text{ is not used later then free}(r_x) \\
& \text{if } r_y \text{ is not used later then free}(r_y) \\
& r_z = \text{allocate}(C) \quad \text{// make a register available for C} \\
& \text{mark } r_z \text{ dirty} \quad \text{// might use } r_x \text{ or } r_y \\
& \text{generate}(op, r_x, r_y, r_z) \quad \text{// emit the actual code}
\end{align*}
\]

for each dirty register \( r \) do :

\[
\text{generate(“move”, r, r \rightarrow opr())}
\]

Cooper/Torczon’s algorithm assumes A,B,C are virtual registers.
We will assume they are variables.
Bottom-Up Register Allocation continued

ensure(opr)
if opr is already in a register \( r \) then
    return \( r \)
else
    \( r = \text{allocate}(\text{opr}) \)
    generate("move", opr, \( r \))
    return \( r \)

allocate(opr)
if there is a free register \( r \) then
    take \( r \)
else
    find \( r \) with the most distant next use
    \( \text{free}(r) \)
    mark \( r \) associated with opr;
    return \( r \)

Next_use analysis:
one backward pass through
the BB is sufficient.
Virtual Registers

A register allocation algorithm can start from two possible situations:

1. All variables are in memory
   • Register allocator needs to worry about aliasing, stores of dirty values, etc. when assigning to physical registers.

2. Variables are placed in virtual registers
   An unlimited number of virtual registers is available.

   Allocation of virtual registers is easy:
   Whenever a new register is needed, a additional register number is taken.
Aliasing and Register Allocation

• Immediately before a load of a variable $x$:
  
  for each variable aliased to $x$ that is on a register association list
  (i.e. that is in a register): **save it**. (so that we are guaranteed to
  load the correct value)

• on store of a variable $x$:
  
  for each variable aliased to $x$ that is on a register association list:
  **remove it** from the list. (so that we will not use a stale value
  later on, i.e. we will reload from memory before use)

• Alias Analyses:
  
  – Most conservative: all variables are aliased
  
  – Less conservative: name-only analysis (no pointers, but
    arrays, in our language)
  
  – Advanced: array subscript analysis, pointer analysis

At subroutine boundaries: often conservative analysis. All (global
and parameter) variables are assumed to be aliased.
Virtual Registers

2. Variables are placed in virtual registers
   An unlimited number of virtual registers is available.
   Allocation of virtual registers is easy:
   • Whenever a new register is needed, an additional register number is taken.
   • Need to worry about aliasing when assigning to virtual registers, not when assigning to physical registers since physical registers will mimic actions taken on virtual registers to cope with dirty values and aliasing
   • What needs to happen is the same as in bullet 1.
   • **The language you are writing your compiler for does not allow aliasing**
The Textbook Algorithm...

• Registers can be:
  – unallocated: carry no value
  – live: carry a value that will be used later
  – dead: carry a value that is no longer needed

• Register association lists:
  variables and temporaries that are associated with a register can be
  – live (L, used again in BB before changed) or dead (D)
  – to be saved (S) at the end of the BB or not to be saved (NS)
    • corresponds to “dirty” attribute in previous algorithm

• Liveness Analysis of Variables:
  – a backwards pass through the code, detecting uses and definitions to determine these attributes.
When to free a register?

- Assume a cost function for register and memory references. E.g., memory ref: 2, register ref: 1
- Freeing costs:
  - 0 (D,NS), (D,S) (no disadvantage in saving right away) D,L dead/live
  - 2 (L,NS) (will need to reload later) S,NS save/no
  - 4 (L,S) (store now, reload later)
- When a register is needed, look for the cheapest. If same cost, free the one with the most distant use, then load the new value and set the status to (L,NS) or (D,NS)
- Assignment to a variable makes previous status (D,NS), i.e. was (L,S), an assign before next use makes it (D,NS)
- This cost may also be used to choose between code generation alternatives, e.g., commutative operations.
- Algorithms on pages 564 .. 566
Register Allocation

An example without optimized register allocation

\[
\begin{align*}
A & := B \times C + D \times E \\
D & := C + (D - B) \\
F & := E + A + C \\
A & := D + E
\end{align*}
\]

\[
\begin{array}{|c|}
\hline
1. (\ast, B, C, T1) \\
2. (\ast, D, E, T2) \\
3. (+, T1, T2, T3) \\
4. (\ast =, T3, A) \\
5. (-, D, B, T4) \\
6. (+, C, T4, T5) \\
7. (\ast =, T5, D) \\
8. (+, E, A, T6) \\
9. (+, T6, C, T7) \\
10. (\ast =, T7, F) \\
11. (+, D, E, T8) \\
12. (\ast =, T8, A) \\
\hline
\end{array}
\]

Load B,R1
* C,R1
Load D,R1
- B,R1
Load E,R1
+ A,R1
Load D,R1
+ E,R1
Store A,R1
- B,R1
Load C,R2
+ A,R1
Load C,R1
+ C,R1
Store F,R1
+ R1,R2
Store D,R2
+ E,R1
Store A,R1
Register Allocation Exercise

Optimized register allocation, textbook, p 568

reduces the cost of storage-to-register and register-to-register operations from 34 to 25

A little explanation of the chart – see regallocbook.pdf in the handouts directory.
Other Register Allocation Schemes

Variations of the presented scheme:
• consider more than one future use
• register “coloring”
• better cost model: consider instruction size and timing; factor in storage-to-register instructions
• include more address modes
• include register-to-register moves, particularly at basic block boundaries
• consider peephole optimizations
Other Register Allocation Schemes

- Register allocation is still a research area.
- As the ratio of memory access time to register access time gets larger, register allocation becomes more important. In some cases may want to use allocator tuned for important application domains.
- One area of research is how to do fast, effective register allocation for Just In Time (JIT) compilers.
- Can use more than one allocation scheme, pick the one that does the best job.
Context-sensitive Code Generation

Generating code from IR trees.

Idea:

- if evaluating R takes more registers than L, it is better to
  - evaluate R
  - save result in a register
  - evaluate L
  - do the (binary) operation

This is because result of R takes a register
Determining Register Needs

Assuming both register-to-register and storage-to-register instructions

For ID nodes (these are leaf nodes):
- left: 1 register
- right: 0 registers (*use op from memory*)

Register need of the combined tree:
\[ X = \]
- \( L+1 \), if \( R = L \)
- \( \max(R, L) \), if \( R \neq L \)
Algorithm for Code Generation Using Register-Need Annotations

Recursive tree algorithm. Each step leaves result in R1 (R1 is the first register in the list of available registers)

Case 1: right branch is an ID:
- generate code for left branch
- generate OP ID,R1 (op,R1,ID,R1)

Case 2: min(L,R) >= max available registers:
- generate code for right branch
- spill R1 into a temporary T
- generate code for left branch
- generate OP T,R1
Tree Code Generation continued

Remaining cases: at least one branch needs fewer registers than available

Case 3: $R < \text{max available registers}$:
- generate code for left branch
- remove first register (R1) from available register list
- generate code for right branch (result in R2)
- generate OP R2,R1

Case 4: $L < \text{max available registers}$:
- temporarily swap R1 and R2
- generate code for right branch
- remove first register (R2) from available register list
- generate code for left branch (result in R1)
- generate OP R2,R1

\[
\min(R,L) < \text{available regs}
\]
Example Tree Code Generation

\[(A-B)+((C+D)+(E*F))\]

<table>
<thead>
<tr>
<th>R1 holds</th>
<th>R2 holds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load C,R2</td>
<td>--</td>
</tr>
<tr>
<td>Add D,R2</td>
<td>--</td>
</tr>
<tr>
<td>Load E,R1</td>
<td>E</td>
</tr>
<tr>
<td>Mult F,R1</td>
<td>E*F</td>
</tr>
<tr>
<td>Add R1,R2</td>
<td>--</td>
</tr>
<tr>
<td>Load A,R1</td>
<td>A</td>
</tr>
<tr>
<td>Sub B,R1</td>
<td>A-B</td>
</tr>
<tr>
<td>Add R2,R1</td>
<td>A-B+C+D+E*F</td>
</tr>
</tbody>
</table>

Note: life gets more interesting if some of the leaves are reused/ across trees